2016 Spring CS300 Homework #2

Due at April 4 (AM) 10:30 on classroom
TA in charge: Soowon Kang (suwon419@kaist.ac.kr)
Reference Textbook: Introduction to Algorithms

1. Analysis of d-ary heaps [Problems for Chapter 6 Heapsort]

A ***d-ary heap*** is like a binary heap, but (with one possible exception) non-leaf nodes have *d* children instead of *2* children.

a. How would you represent a ***d-ary*** heap in an array?

b. What is the height of a ***d-ary*** heap of *n* elements in terms of *d* and *n*?

c. Give an efficient implementation of ***EXTRACT-MAX*** in a ***d-ary*** max-heap. Analyze its running time in terms of *d* and *n*.

d. Give an efficient implementation of ***INSERT*** in a ***d-ary*** max-heap. Analyze its running time in terms of *d* and *n*.

e. Give an efficient implementation of ***INCREASE-KEY(A, i, k)***, which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the ***d-ary*** max- heap structure appropriately. Analyze its running time in terms of *d* and *n*.

2. Stack depth for quicksort **[Exercises and Problems for Chapter 7 Quicksort]**

The following procedure implements quicksort:

```
QUICKSORT (A, p, r)
1 if p < r
2       q = PARTITION (A, p, r)
3       QUICKSORT (A, p, q − 1)
4       QUICKSORT (A, q + 1, r)
```

The key to the algorithm is the *PARTITION* procedure, which rearranges the subarray *A[p ... r]* in place.

```
PARTITION (A, p, r)
1 x = A[r]
2 i = p − 1
3 for k = p to r − 1
4       if A[k] ≤ x
5               i = i + 1
6               exchange A[i] with A[k]
7 exchange A[i+1] with A[r]
8 return i + 1
```

The *QUICKSORT* algorithm contains two recursive calls to itself. After *QUICKSORT* calls *PARTITION*, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in *QUICKSORT* is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT (A, p, r)
1       while p < r
2               // Partition and sort left subarray.
3               q = PARTITION (A, p, r)
4               TAIL-RECURSIVE-QUICKSORT (A, p, q − 1)
5               p = q + 1
```

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

a. What value of q does ***PARTITION*** return when all elements in the array $A[p \ldots r]$ have the same value? Modify ***PARTITION*** so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p \ldots r]$ have the same value.

b. Show that the running time of ***QUICKSORT*** is $\boldsymbol{\Theta(n^2)}$ when the array A contains distinct elements and is sorted in decreasing order.

c. Argue that ***TAIL-RECURSIVE-QUICKSORT*** (A, 1, $A.length$) correctly sorts the array A.

d. Describe a scenario in which ***TAIL-RECURSIVE-QUICKSORT***'s stack depth is $\boldsymbol{\Theta(n)}$ on an n-element input array.

e. Modify the code for ***TAIL-RECURSIVE-QUICKSORT*** so that the worst-case stack depth is $\boldsymbol{\Theta(\lg n)}$ Maintain the $\boldsymbol{O(n \lg n)}$ expected running time of the algorithm. (**lg** is the binary logarithm which uses base 2)